O’Caml

by
Pavan Krishnamurthy and Qiyam Tung

Table of Contents

Introduction
OCaml is the most popular variant of the ML branch Caml (Categorical Abstract Machine Language). It is a functional programming language with an added advantage of type inference and static typing, which means that most types are safe and there are very less chances of run time failures in OCaml. So it is considered a safe language. There are lot of intresting features in OCaml which attracts users and makes it an efficient functional language.

History of ML/OCaml
ML was designed by Robin Milner to allow programmers to develop theorem proving techniques. A series of events led to the development of Caml, which was based on a branch of mathematics developed by Pierre-Louis Curien that allowed for the techniques to be proven in a simple way. In the 90's, object-oriented programming and type systems were a hot research area. The hype led to the development of OCaml, which is object-oriented but elegantly done, unlike the patched C++. According to the website, OCaml "was the first language (and still is the only language) that combines the full power of object-oriented programming with ML-style static typing and type inference."

Audience
While OCaml is an impure functional programming language, it distinguishes itself from most other functional languages by being fast. Based on benchmarks, on average, OCaml performs at least 50% the speed of equivalent C programs, which is fairly good. One of the reasons for this performance is that it was built to be fast from the beginning; furthermore, it supports imperative features in cases where functional programming cannot provide the speed necessary for the application.
A distinct feature of OCaml and of the ML programming languages is their use of static types and type inference. This combination allows the compiler to prevent programming errors one usually encounters in Java and C/C++. That is, it guarantees runtime safety without runtime checks (except for cases such as array bounds checking).
It is an attractive programming language in that it is has the elegance and readability of functional languages, acceptable execution time, as well as the ability to write large programs thanks to its object-oriented features. It is more tailored towards more numerical programming and in creating programs the assist in proofs. For example, *Coq Proof Assistant*, as its name implies, helps to verify mathematical proofs.

The following are features of OCaml are:

Types

- *Type Inference*
  OCaml is a static-typed language. This means it picks up the type mismatch at the compile time. This makes the interpreter work easier when executing it. Static typing also makes the programs safer , since everything is checked at compile time they are proven to run safe. A dynamically typed language has the advantage of not needing to define its type during compile-time, but potential type errors could occur during runtime. Along with static typing OCaml is also type inferred language which means the user need not specify what type of variable it is , it's the compiler job to detect it. For example.

  ```
  let multiply2 x = x * 2;;
  ```

  Since 2 is multiplied with x , compiler infers that x has to be a number and it later verifies by the operator specified. Because of the potential ambiguity that arises from floating

point and integers, ***.** is used to denote floating point multiplication. Hence if instead of 2 it is 2.0 compiler would have generated error saying wrong operator. Hence this function is

```
val multiply2 : int -> int = <fun>
```

Furthermore, if the use of the type is rather ambiguous, then it automatically converts it to a generic type. For example, if a method were defined to return the size of a list, OCaml would allow this method to work on lists of any type.

```
# let rec length = function
      | ([]) -> 0
      | (x::xs) -> 1 + length(xs);;
val length : 'a list -> int = <fun>
```

Naturally, there may be cases where the programmer may want to limit the function to a specific type. OCaml also allows the user to explicitly declare the type of the value.

- *Simple Types*
  The usual types are offered in OCaml, such as integers, floating-point numbers, Booleans, characters,
  and strings. Additionally, OCaml offers tuples, arrays, and lists. As with most functional languages, most of the types in OCaml are immutable. Exceptions include arrays, references, and records (which can be mutable and immutable).
  Tuples are simple structures that resemble structs in C. It can hold an arbitrary number of items as well as different types. Unlike structs, a definition does not need to be explicitly defined for a tuple instantiation.

```
# let value = (1,"Give us", 3.5, ("an A", '!'));;
val value : int * string * float * (string * char) =
(1, "Give us", 3.5, ("an A", '!'))
```

  Lists are homogeneous data structures that can hold multiples instances of the same type. It is recursively defined. That is, a list is composed of a head (the first element's value) and a tail (the rest of the list). The following code demonstrates this.

```
val aList : int list = [1; 2; 3]
# List.hd(aList);;
- : int = 1
# List.tl(aList);;
- : int list = [2; 3]
```

  Arrays are the same as lists with the difference being that it has mutable data (as noted above).

Labels

Labels allow the compiler to do more checking and allows for better documentation. For example, in a normal function:

```
# String.sub;;
-  : string -> int -> int -> string = <fun>
```

```
-
```

It is difficult to tell what the function arguments are. With labels, it becomes clearer:

```
# StringLabels.sub;;
-  : string -> pos:int -> len:int -> string = <fun>
-
```

This also allows the function to take in arguments out of order so long as the argument is prefixed with the label. When the function arguments are preceeded by ~ , it means they can be referred by their names. This definitely adds an advantage as the user need not remember the order of arguments, but instead the argument name would suffice.

```
# let divide ~numerator ~denominator =
numerator/denominator;;
val divide : numerator:int -> denominator:int -> int =
<fun>
# divide ~numerator:13 ~denominator:3;;
-  : int = 4
-
# divide ~denominator:40 ~numerator:80;;
-  : int = 2
```

As a result, you can now curry functions in a useful way. In a function that divides two numbers, without labels it would not be possible to specify the divisor. Rather, the programmer could only create an alias that has a default dividend. In this example, we show that with labels, specifying a dividend is not just a dream.

```
#let divideByFive = divide ~denominator:5;;
val divideByFive : numerator:int -> int = <fun>
# divideByFive 25;;
- : int = 5
```

Variants

Variants are like C unions, but a lot safer which means you cannot assign values to wrong types ,doing which the compiler throws the error.

```
type major = Nothing | Communications of string |
Mathematics of int;;
```

which means major can be nothing or Communications, or a Mathematics type

```
#let value = Mathematics 3;;
val value : major = Mathematics 3
```

- *Recursive variants (trees)*

```
type LinkedList = Value of int | Next of LinkedList ;;
```

- *Parameterized variants*

```
type 'a LinkedList = Value of 'a | NEXT of 'a
LinkedList;;
```

Unlike C, you cannot get conflicts:

```
type engine = On | Off;;
```

```
type lights = On | Off;;
```

Clearly, the question is what type does `Open belong to now`?

```
# On;;
: lights = On
```

So OCaml chooses whichever is latest definition to be the type object, this is the beauty of type inference and variants.

Aliases (Currying)
OCaml allows you to define aliases for function names and arguments.

```
let multiply x y = x * y;;
let multiply2 = multiply 2;;
```

The combination of the small probability of side effects in functional programming with aliases (or currying) allows for powerful modular programming.

Pattern Matching
While this is not strictly a feature of functional programming languages, it is still a common and useful feature in functional languages.

Basic pattern matching in functions:

```
let sum = function
  | [] -> 0
  | x::xs -> x + sum xs;;

val sum : int list -> int = <fun>

let rec mapNumToList = function
   | (_, []) -> []
   | (num, x::xs) -> num*x::mapNumToList (num, xs);;

val mapNumToList : int * int list -> int list = <fun>
```

You can also take in a tuple as an argument. _ means unit, which means that the value is irrelevant in the particular matching. For example, in the first match we see that if it's an empty list, we don't need to multiply its elements by the value. This saves binding a name to the value. OCaml also allows the programmer to use the keyword `match` to match to a specific argument. Thus, here is an example that uses cons and pattern matching that allows the programmer to sort.

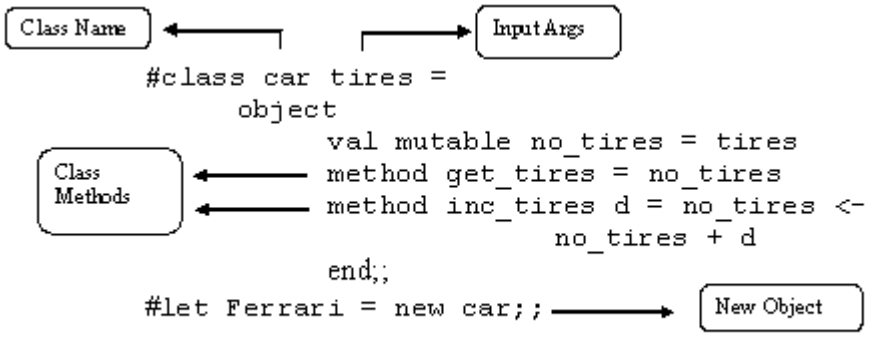
```
let rec sort lst =
   match lst with
     [] -> []
   | head :: tail -> insert head (sort tail)
 and insert elt lst =
   match lst with
     [] -> [elt]
```

```
   | head :: tail -> if elt <= head then elt :: lst else
head :: insert elt tail
 ;;
```

Finally, as OCaml also allows symbolic processing. As a result, with variants and pattern matching, OCaml can process symbolic algebraic expressions.

Object Oriented Features

- *Classes & Objects*
  OCaml integrates ML with Object oriented concepts to give a powerful Object Oriented language. Like all object oriented languages it supports classes and objects. Classes are easy to define and use.



  Mutable variables are used because the value of *no_tires* is updated in the inc_tires method. This is how a class and object is defined in OCaml.

- *Immediate Objects*
  Immediate objects are a way to create an object without declaring a class for it. This kind of declaring an object can be useful to define simple objects and it avoids unnecessary steps to declare a class and then creating an object for it. For example

```
#let findmax x y =
      if x > y then object method max = x end
      else object method max = y end;;
```

  The function findmax takes two arguments x & y and returns an object with a method max in it. Its an easy way to define objects, this kind of defining an object is useful for small objects.

- *Functional Objects*
  Functional Objects also called functors are similar to function pointer in C except that it will be pointing to an object . The construct {<..>} returns a new SELF object but the state of the object would be after executing the statements between the two angled brackets .

```
#class car tires =
object
      val mutable no_tires =  tires
      method get_tires = no_tires
      method inc_tires d ={< no_tires = no_tires + d >}
end;;
```

In this example when call to inc_tires is done by any object , the method returns the new object of type car but the value of no_tires would be have been incremented by d. This combined with parametric polymorphism makes generics more powerful..

- *Cloning Objects*
  OCaml also supports Object cloning, means to make a shallow copy of the object. When object is cloned it returns a new object which is same as the old object. The values of instance variables will be copied to the new object, in other words an object with same state is created. Ocaml achieves this by using **Oo.copy** which creates a clone of the original object.

```
#let bmw = Oo.copy Ferrari;;
```

  This creates a bmw object a clone of Ferrari.

Inheritance

Ocaml does support inheritance and along with it also has multiple inheritance. For example

```
#class Ferrari
inherit car as super
.........
end;;
```

This is just a single inheritance , now to define multiple inheritance the inherit statement is specified twice. The thing to be noted is if the classes which are inherited have common methods then only the last inherited class method is kept. However the other class methods can be accessed by typecasting the object. This avoids lot of confusion which exists in c++ multiple inheritance.

```
#class BMW
inherit car as acar                 } Multiple Inheritance
inherit Ferrari as aferrari         }
.....
end;;
```

Coercions

Coercions are similar to casting in C++ and Java though there are enough differences to validate a different name. The most important aspect of a coercion is that it is a semantic (as opposed to syntactic) relationship between object. That is, it is not equivalent to inheritance. Second, unlike C++ and Java, there is no downcasting, or narrowing. Since adding such a feature would add runtime checking, OCaml rids themselves of this slow but at times useful feature. Finally, subtyping is never implicit.

```
# class vehicle (nm : string) =
    object
      val name = nm
      method getName = name
    end;;
class vehicle :
  string -> object val name : string method getName :
string end
# class car nm (wls: int) =
      object
      inherit vehicle nm
      val numWheels = wls
      method getNumWheels = wls
```

```
    end;;
class car :
  string ->
  int ->
  object
    val name : string
    val numWheels : int
    method getName : string
    method getNumWheels : int
  end
# let aCar = new car "Honda" 3;;
val aCar : car = <obj>
# let aMethod (a : vehicle) = a#getName;;
val aMethod : vehicle -> string = <fun>
# aMethod aVehicle;;
- : string = "hiccup"
# aMethod (aCar : car :> vehicle);;
- : string = "Honda"
```

As this example shows, by coercing our car object into a vehicle, we are able to call the method with `aCar`. However, without coercing the object,

```
#aMethod aCar;;
```

This expression has type car but is here used with type vehicle. Only the first object type has a method getNumWheels. OCaml will give an error as subtyping must be explicit. Furthermore, as noted before, the class does not necessarily have to inherit from vehicle.

```
# class thing =
    object
     method getName = "something"
    end;;
class thing : object method getName : string end
# let aThing = new thing;;
val aThing : thing = <obj>
# aMethod (aThing : thing :> vehicle);;
- : string = "something"
```

Clearly, this is not the same as inheritance. However, without implicit subtyping, there is no need to worry about possible incompatibility. Since the programmer must explicitly declare the type to a subtype of some class, the responsibility lies in the programmer. This feature contrasts with OCaml's decision to use type inference to reduce bloated code, for better or for worse.

Parametric Polymorphism

OCaml is also a language which supports parametric polymorphism. As C++/ Java term parametric polymorphism term them as generics OCaml has its own representation `(').` For example

```
#let append value list = value::list;;
val append : 'a -> 'a list -> 'a list = <fun>
```

which means function append takes a value , a list and returns a list. The **'a** denotes a particular type. It can be an int, float, string, etc. so long as list homogeneous.

Classes can take generic arguments. However, the OCaml requires that the programmer explicitly parameterize the class. Unlike simple methods, OCaml class methods require that the type be a specific type (whether the parametric type or a specific type). For example

```
# class ['a] aClass init =
    object
      val mutable x = (init: 'a)
      method get = x
      method set y = x <- y
    end;;

     class ['a] aClass :
  'a -> object val mutable x : 'a method get : 'a method
set : 'a -> unit end
```

Note that argument y in set is inferred to be ‘a. As the class is parameterized, it cannot simply accept an argument of any type.

```
# class ['a] Class init =
      object
        val mutable x = (init: 'a)
        method get = x
        method return y = y
      end;;
Some type variables are unbound in this type:
  class ['a] aClass :
    'a ->
    object val mutable x : 'a method get : 'a method return
: 'b -> 'b end
The method return has type 'a -> 'a where 'a is unbound
```

Exception Handling

OCaml supports exception handling . Similar to try catch block in java/ c++ OCaml has **try-with block** for catching exceptions. Exceptions uses Patter matching to find if a particular exception is matched, if it is it raises the exception. Exceptions can also be created by users using e*xception:*

```
exception Key_not_Found
```

and it can be thrown by *raise* keyword

```
raise Key_not_Found.
```

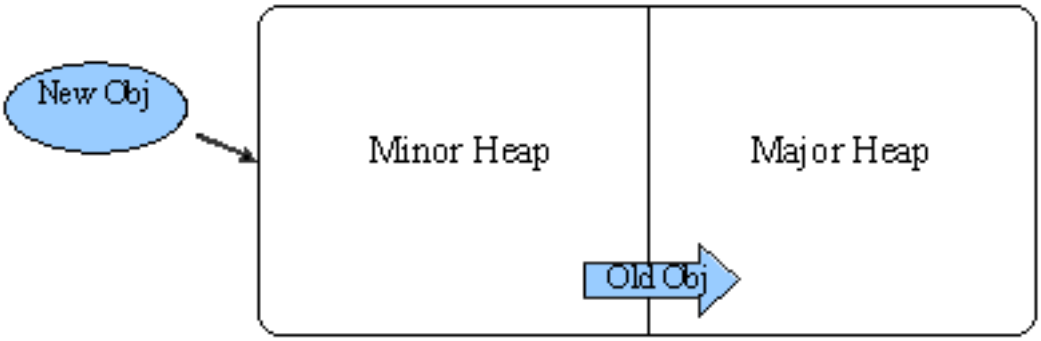
There are some predefined exceptions which OCaml has defined *Not_Found, Failure, Invalid_argument and End_of_File*

Garbage Collector

OCaml follows **Generational collection** approach for Garbage collection. The collector divides the heap into two generation namely Major and Minor, this recognizes a general principle: Most objects are small and allocated frequently and then immediately freed. These objects go into the minor heap first, which is garbage collected frequently. Only some objects are long lasting. These objects get promoted from the minor heap to the major heap after some time, and the major heap is only collected infrequently.

The Garbage collection does not run in separate thread, it gets called when there is in sufficient memory and when new is called on an object. It’s a synchronous activity so new stops till garbage

collection is done. But it doesn't have to stop for so long since number of live objects in Minor heap would be less and will thus garbage collected faster.



There are some interesting features of Garbage collection , such as OCaml provides special function to printout the statistics of the heap . It would print what is the size of major and minor heap , how much of size is free in the heap , the number of fragments and how much of compaction was performed. This can be effectively utilized to analyze the garbage collector performance and programs can be tuned for better performance.

```
Gc.print_stat stdout;;
```

The above statement is the function call to print the statistics to the standard output. One more special function is to set the debugging messages to print when some operations are performed by garbage collector like on every major heap collection , or when objects are moved from minor heap to major heap.

```
Gc.set { (Gc.get ()) with Gc.verbose = 0x01 };;
```

The above statement sets the debugging message to print whenever a major heap collection is performed. OCaml also provides users to write finaliser method for an object which gets called when the object is about to garbage collected. Ocaml has Weak pointers , which when set to an object the programmer tells the garbage collector that it can garbage collect the object at any instant irrespective of whether it is live or dead. This kind of pointer would be useful for temporary objects collecting which the garbage collector need not waste time.

OCaml Intrepreter

OCaml similar to Java has its interpreter to execute its byte code. The OCaml compiler converts the source file to an platform independent intermediate byte code representation later which the interpreter executes it or there is also an option of converting it to native code which runs faster but its not portable. OCaml compiler allows incremental compilation which means when there are large number of compilation units and if the user changes only one it , then it is waste to compile all the units but instead OCaml compiles only the modified unit and then links with the rest. This option of compilation saves precious compilation time and provides faster byte code generation. The command to invoke byte code compiler is *ocamlc* , while the native code compiler can be invoked by *ocamlopt.*